Sep 97

A Hybrid Immersive / Non-Immersive Virtual Environment Workstation

N96-057
Department of the Navy

Report Number 97265

Submitted by:
Fakespace, Inc.
241 Polaris Ave.
Mountain View, CA 94043
Phone (415) 688-1940
Fax (415) 688-1949

Introduction

The Hybrid Immersive / Non-Immersive Virtual Environment workstation integrates two different approaches to the presentation and exploration of virtual spaces. The Immersive and Non-Immersive components of the system are complimentary in that each approach provides a distinct perspective on the virtual environment being examined.

Navigation with the Immersive Workstation

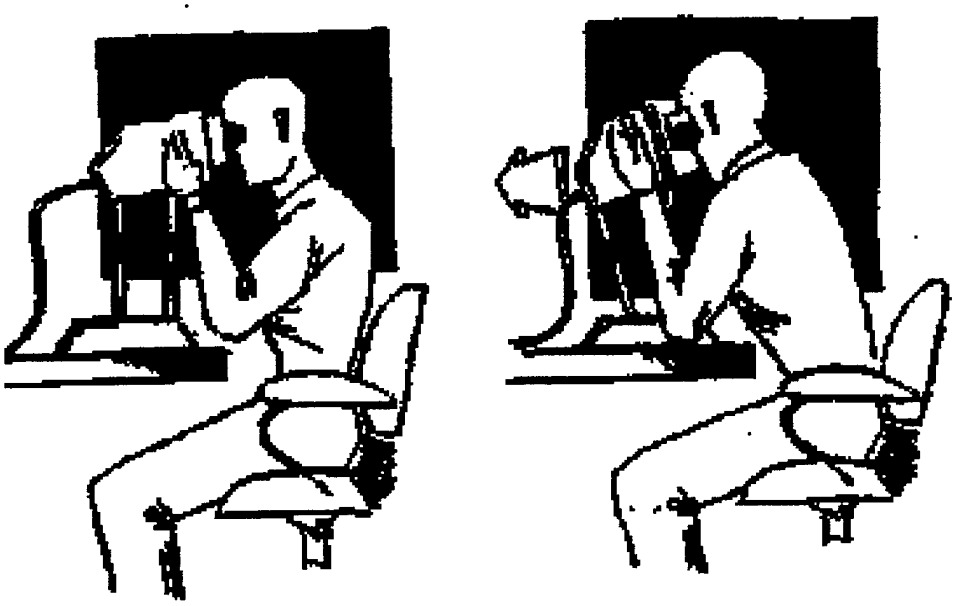
The Immersive workstation provides an interface which permits the user to move around in the virtual environment in a unique way. The PUSH allows the operator move around using natural body motion which is mapped to motion in the virtual environment.

In daily life when we want to see something, we just move our bodies around by walking, bending, turning and so forth until we can see it. It is by this naturally coordinated movement that we change our present viewpoint to some new, desired viewpoint. While we have very sophisticated and naturally integrated human hardware for dealing with the physical world, we have no such natural interface to support behavior in virtual environments.

The PUSH interface technology provides a means of yoking and measuring the exertion of orienting muscle groups while constraining the actual movement of those muscles. The muscles are those having either their origins and or insertions on the spinous and or transverse processes of the vertebral column. They also include some muscles that have their insertions on the calvarium and on the pelvic bones. These muscles act to produce head-movement and thus, control viewpoint, in the three rotational degrees of freedom. The mechanical yoking of muscle groups to a compliant mechanical linkage and sensor mechanism provides force feedback information for control of one's viewpoint without requiring full physical motion from the user. The PUSH maps this exertion to the rate and direction of movement in the virtual environment.

The advantages and working of the PUSH technology can be better explained and understood by the following consideration of three neuroanatomical and structural principles of the body.

Firstly, the vertebral column is substantially fluid and flexible. Generally speaking, the intersection of a rotation plane and its axis of rotation can occur nearly anywhere along the vertebral column from the lumbar area up to the base of the skull. For example, when humans plan to orient towards an object of interest, this rotation could occur at the neck or it could occur at the level of the shoulders while keeping the neck stiff or at the level of the hips while keeping the neck and shoulders stiff or some combination of these movements. The same is true of pitching and rolling motions.



Secondly, the muscles which move and orient the head and trunk send special sensory information to the brain while they are contracting. This special sensory information permits knowledge of one's orientation independent of other sensory cues such as sight or balance. Therefore, simply by the action of the muscles that orient the head and trunk we can know when we have substantially achieved an orientation. There are a number of physical illusions which illustrate this.

Lastly, voluntary muscular coordination is driven by our perception of achieving a goal of a muscular movement. This means we tend to exert as much force as required, but only enough force to efficiently reach our desired end-point. This has been called the law of minimal spurt action.

The PUSH capitalizes on these three principles. Because of the fluidity of the vertebral column, the device effectively yokes the movement of the orienting muscles in the physical world to movement in the virtual environment without the disadvantages of a head-mounted display. Therefore, to orient toward an area of interest in the leftward portion of the virtual world, the user simply and naturally activates the muscles that would be used to orient left in the physical world. At the same time, the effect of virtual motion toward a goal is compelling because while the muscles are contracting, there is special sensory information sent to the brain telling it that such movement should be occurring. This movement is confirmed by the smooth flow of images in the display depicting movement toward the goal until the area of interest is in view. Hence the perception of natural and intuitive bodily movement has occurred even though only constrained bodily movement has actually taken place.

For example, the user may be seated looking into the PUSH with two hands grasping the display which is also the navigational input device. The arms are used to forcibly interact with the PUSH using the orienting muscles of neck and trunk. This force information is fed to the computer where the user's motion in the virtual world is controlled. In this way, the user manipulates the area of interest so that if, for example, the user physically turns left the viewpoint in the virtual world is manipulated appropriately. When the user sees what is desired,

that is, reaches the area of interest, these muscles begin to relax and the virtual motion is accordingly slowed to a standstill.

The core concept behind the PUSH is the mapping of the exertion of the orienting muscle groups with an appropriate and expected rate of change in one's viewpoint in the virtual environment. We have been impressed with effectiveness of this approach to pair muscle exertion with apparent virtual motion thereby providing free movement in a virtual world without having to leave one's chair.

Navigation with the Non-Immersive Workstation

The PUSH provides an intuitive and natural method for the navigation and exploration of virtual worlds in an immersive environment. Indeed, the display and the PUSH input device are the same physical unit. In order to present a unified navigational interface, we have experimented with using an alternative form to accomplish the same bi-manual interface with the Non-Immersive display system.

The PushStick is an input device which has similar characteristics to the desktop PUSH. The device itself is a full six degree of freedom controller. The device is grabbed as shown below.

The PushStick interfaces to the Silicon Graphics Workstation via a serial port. The navigational interface is consistent with that of the Desktop PUSH. In order to move in a particular direction, one simply pushes the device in that direction. Depending on the sensitivity of the device and the amount you want to move, it may be controlled either with your hand or with your finger tips. The bi-manual control and mechanical design of the device make it relatively easy to control more than one degree of freedom at once. Moving forward/backwards, left/right and up/down are easily accomplished by moving both hands equally in the direction one would like to move. Rotations are accomplished in a similar fashion by moving the hands differentially. Thus, to look to the right, the right hand is pulled towards the body and the left hand extended. The real advantage of the bi-manual control becomes apparent when one wishes to perform motions which do not correspond to one of the principle axes. For example, moving in an arc around an object at some distance is a coordinated move which involves moving a fraction to the right, then twisting a fraction to the left. Coordinated moves of this type can be accomplished at any scale and do not require any object selection or mode changes. The shape of this arc is defined by one's body exerting the muscles to achieve a goal and after very little practice it is easy to orbit around arbitrary objects. The visual feedback from the display system makes the fine tuning of the control loop relatively easy. This interface provides a consistent navigational paradigm between the Immersive and Non-Immersive systems.

In addition to providing a navigational interface that is well suited to performing motions which are not necessarily aligned to the axes of the device, the interface also provides a high degree of control over the scale of motions. The device is mapped such that small force exertions correspond to very small motions. Larger force exertions map to very much more significant motions. The non-linear mapping between exerted force and resulting motion corresponds to the logarithmic response of the axial muscles.

Appendix 1 - Software interface for the PushStick

```
/* PushStick interface software
 * Handles interface from the device to shared memory.
 * Shared memory data is used in the Standard VLIB software
 * in the same routines as the PUSH.  Thus the PushStick uses
 * the same non-linear variables as defined in the desktop.config
 * files.
 *
 * IEM 5/97
 * Code based in large part on the software for the
 * Logitech Magellan which is used as the sensor in
 * the PushStick.  The sensor is coupled to the handles
 * with a unique mechanical interface which allows free
 * and fluid motion of the handles while not over
 * stressing the sensor.
 */
#include <stdio.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <ctype.h>
#include <termio.h>
#include <termios.h>
#include <fcntl.h>
#include <sys/param.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

#include <string.h>
#include "shmlib.h"

#define TRUE        (1==1)
#define FALSE               (1==0)

typedef unsigned int  word;

#define Display           FALSE

struct _pskData {
 signed int x;
 signed int y;
 signed int z;
 signed int a;
 signed int b;
 signed int c;
};
typedef struct _pskData pskData;

char NibbleTable[] = "0AB3D56GH9:K<MN?";

int pskNibbleError = FALSE;

signed int pskNibble( char Value )
{
int value;

 value = Value & 0x000F;
 if ( NibbleTable[ value ] == Value )
  return value;
 pskNibbleError = TRUE;
 printf( "\n\r Nibble Error %02X %02X %d \n\r", Value, NibbleTable[value],
value );
 return value;
```

```
}

int pskHandle = -1;

char pskLineBuffer[64];

int pskRead( int pskHandle, char * pskBuffer, int pskLength )
{
char pskLocalBuffer[32];
char *pskReturn;

 do
  {
   pskReturn = strchr( pskLineBuffer, '\r' );
   if ( pskReturn == NULL )
   {
    memset( pskLocalBuffer, 0x00, sizeof( pskLocalBuffer) );
    read( pskHandle, pskLocalBuffer, sizeof( pskLocalBuffer ) );
    strcat( pskLineBuffer, pskLocalBuffer );
    continue;
   }
  else
   {
    *pskReturn = '\000';
    if ( strlen( pskLineBuffer ) > pskLength )
     {
      printf( "Receive Buffer overflow!\n\r" );
     }
    else
     {
      strcpy( pskBuffer, pskLineBuffer );
      strcat( pskBuffer, "\r" );
      strcpy( pskLineBuffer, ++pskReturn );
      return strlen( pskBuffer );
     };
   };
  } while ( TRUE );
}

char pskVersionString[1024];

#define pskVersionText    "vQ\r"

char * pskVersion( )
{
char pskBuffer[128];

 write( pskHandle, pskVersionText, strlen(pskVersionText) );
 do
  {
   memset( pskVersionString, 0x00, sizeof( pskVersionString ) );
   pskRead( pskHandle, pskVersionString, sizeof( pskVersionString ) );
  } while ( pskVersionString[0] != 'v' );
 return pskVersionString;
}


#define pskModeOff  "m0\r"       /* Translation and Rotation Mode OFF */
#define pskMode           "m3\r"       /* Translation and Rotation Mode ON */
#define pskDataRate "p00\r"            /* 100 msec Date Rate */
#define pskQuality  "q00\r"            /* Linear */
#define pskNullRadius     "n?\r"       /* Null Radius at 5 */
#define pskBeeper   "b9\r"       /* Short Beep */
#define pskZero           "z\r"        /* Zeroing */
#define pskDataQuestion   "dQ\r"       /* Data Request (Polling mode) */
```

```
#define pskStartup  "\rz\rz\r"

int pskStarter()
{
char pskBuffer[128];
int loop;

 memset( pskLineBuffer, 0x00, sizeof( pskLineBuffer ) );
 write( pskHandle, pskStartup, strlen( pskStartup ) );
 pskRead ( pskHandle, pskBuffer, sizeof( pskBuffer ) );
 if ( Display )
  {
   for ( loop=0; pskBuffer[loop]!='\000'; ++loop )
    printf( "%02X ", pskBuffer[loop] );
   printf( "\n\r" );
  };
}

int pskTranpskitReceive( char * StringIn, char *StringCompare )
{
char pskBuffer[128];

 write( pskHandle, StringIn, strlen(StringIn) );
 if ( StringIn[0] == '\r' )
  return TRUE;
 if ( Display )
  printf ("Tranpskit: %s\n", StringIn );
 do
  {
   memset( pskBuffer, 0x00, sizeof( pskBuffer ) );
   pskRead( pskHandle, pskBuffer, sizeof( pskBuffer ) );
   if ( Display )
    printf ( "Receive: %s\n", pskBuffer );
  } while ( StringIn[0] != pskBuffer[0] );
 return strcmp( pskBuffer, StringCompare ) == 0;
}


int main( int argc, char *argv[] )
{
word Keyboard = 0;
pskData pskMemData;
struct termio termio;
int len, result, loop, End;
char pskBuffer[128], pskRS232[128];
shared_mem_if *buf = (shared_mem_if *)map_shared();

       buf->in_use = 0;


 if ( argc == 2 )
  {
   if ( atoi( argv[1] ) > 0 )
    sprintf( pskRS232, "/dev/ttyd%d", atoi( argv[1] ) );
  }
 else
  sprintf( pskRS232, "/dev/ttyd%d", 3 );
 printf("Use RS232-Device %s\n\r", pskRS232 );

 pskHandle = open( pskRS232, O_RDWR );
 if ( pskHandle == -1 )
  {
   printf( "Can't open seriel port\n" );
   return -1;
  };

 result = ioctl( pskHandle, TCGETA, &termio );
```

```
termio.c_iflag = IGNBRK | IGNPAR;
termio.c_oflag = 0;
termio.c_cflag = B9600 | CREAD | CS8 | CSTOPB | HUPCL; /* CLOCAL */
termio.c_lflag = 0;                                    /* -ECHO */
termio.c_line = LDISC1;
termio.c_cc[VEOL] = '\r';
termio.c_cc[VERASE] = '\000';
termio.c_cc[VKILL] = '\000';
termio.c_cc[VMIN ] = 1;
termio.c_cc[VTIME] = 0;


result = ioctl( pskHandle, TCSETA, &termio );
result = ioctl( pskHandle, TCGETA, &termio );

sleep( 1 );

pskStarter();

while ( ! pskTranpskitReceive( pskDataRate, pskDataRate ) ) ;

pskVersion( );

while ( ! pskTranpskitReceive( pskQuality, pskQuality ) ) ;

while ( ! pskTranpskitReceive( pskNullRadius, pskNullRadius ) ) ;

while ( ! pskTranpskitReceive( pskZero, pskZero ) ) ;

while ( ! pskTranpskitReceive( pskBeeper, "b\r" ) ) ;

while ( ! pskTranpskitReceive( pskMode, pskMode ) ) ;

printf( "\n\n%s\n\n", &pskVersionString[1] );
write ( pskHandle, pskDataQuestion, strlen( pskDataQuestion ) );

End = FALSE;

while ( !End )
 {
  memset( pskBuffer, 0x00, sizeof( pskBuffer ) );

  pskRead ( pskHandle, pskBuffer, sizeof( pskBuffer ) );

  len = strlen( pskBuffer );

  if ( pskBuffer[0] == 'k' )
   if ( strlen( pskBuffer ) == 5 )
    {
     Keyboard = pskNibble( pskBuffer[1] ) +
             pskNibble( pskBuffer[2] )*16 +
             pskNibble( pskBuffer[3] )*256;
     End = Keyboard == 0x0080;
    };

  if ( pskBuffer[0] == 'd' )
   if ( strlen( pskBuffer ) == 26 )
    {
     pskNibbleError = FALSE;
     pskMemData.x =          ( pskNibble( pskBuffer[1] ) * 4096 +
                         pskNibble( pskBuffer[2] ) * 256 +
                         pskNibble( pskBuffer[3] ) * 16 +
                         pskNibble( pskBuffer[4] ) - 32768 );

     pskMemData.y =          ( pskNibble( pskBuffer[5] ) * 4096 +
                         pskNibble( pskBuffer[6] ) * 256 +
```

```
                        pskNibble( pskBuffer[7] ) * 16 +
                        pskNibble( pskBuffer[8] ) - 32768 );

      pskMemData.z =          ( pskNibble( pskBuffer[9] ) * 4096 +
                        pskNibble( pskBuffer[10] ) * 256 +
                        pskNibble( pskBuffer[11] ) * 16 +
                        pskNibble( pskBuffer[12] ) - 32768 );

      pskMemData.a =          ( pskNibble( pskBuffer[13] ) * 4096 +
                        pskNibble( pskBuffer[14] ) * 256 +
                        pskNibble( pskBuffer[15] ) * 16 +
                        pskNibble( pskBuffer[16] ) - 32768 );

      pskMemData.b =          ( pskNibble( pskBuffer[17] ) * 4096 +
                        pskNibble( pskBuffer[18] ) * 256 +
                        pskNibble( pskBuffer[19] ) * 16 +
                        pskNibble( pskBuffer[20] ) - 32768 );

      pskMemData.c =          ( pskNibble( pskBuffer[21] ) * 4096 +
                        pskNibble( pskBuffer[22] ) * 256 +
                        pskNibble( pskBuffer[23] ) * 16 +
                        pskNibble( pskBuffer[24] ) - 32768 );

      if ( pskNibbleError )
       memset( &pskMemData, 0x00, sizeof( pskMemData ) );
     };

    switch( pskBuffer[0] )
     {
      case 'd' :
      case 'k' :
      case 'm' :
      case 'n' :
      case 'z' :
      case 'b' :
      case 'q' :
      case 'p' :
      case 'v' :
             while(buf->in_use);
             buf->in_use++;
                   buf->pos[0] = (float) pskMemData.x;
                   buf->pos[1] = (float) pskMemData.y;
                   buf->pos[2] = 0.5 * (float) pskMemData.z;
                   buf->ori[0] = 0.5 * (float) pskMemData.a; /* tilt */
                   buf->ori[1] = (float) pskMemData.b;
                   buf->ori[2] = (float) pskMemData.c;
             buf->in_use--;

#if 1
      printf( "%02X %c  x=%6d y=%6d z=%6d a=%6d b=%6d c=%6d
%c%c%c%c%c%c%c%c %s\r",
             pskBuffer[0],  pskBuffer[0],
             pskMemData.x, pskMemData.y, pskMemData.z,
             pskMemData.a, pskMemData.b, pskMemData.c,
             Keyboard & 0x0001 ? '1':'-',
             Keyboard & 0x0002 ? '2':'-',
             Keyboard & 0x0004 ? '3':'-',
             Keyboard & 0x0008 ? '4':'-',
             Keyboard & 0x0010 ? '5':'-',
             Keyboard & 0x0020 ? '6':'-',
             Keyboard & 0x0040 ? '7':'-',
             Keyboard & 0x0080 ? '8':'-',
             Keyboard & 0x0100 ? '*':'-' , "");
        fflush( stdout );
#endif
      break;
```

```
    };
  };

 while ( ! pskTranpskitReceive( pskModeOff, pskModeOff ) )
  ;
 printf( "\n\r\n\rEnd\n" );

 close( pskHandle );
 return 1;
}
```